REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-01-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden to Department of Defense, Washington Headquarters Services Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*
03-2001

2. REPORT TYPE
Professional Paper

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
On the Role of Randomization in Software Engineering

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

5d. PROJECT NUMBER
ZA01

5e. TASK NUMBER

5f. WORK UNIT NUMBER

6. AUTHORS
Rubin, S. H. (1)
Trajkovic, L. (2)
Boerke, J. (1)
Rush, R. J., Jr. (1)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
1) Space and Naval Warfare Systems Center
53560 Hull Street
San Diego, CA 92152-5001

2) Simon Fraser University
Vancouver, BC Canada

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

20011217 279

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Randomization is defined to mean the removal of redundancy from information. In this sense, it is synonymous with information compression; although, randomization may extend beyond syntactic representation to include domain-specific semantic elements as well. This paper serves to make clear the ubiquitous role assumed by randomization in all aspects of software engineering – from programming language design to program design to testing. It goes on to show that the representation of knowledge in what is termed an expert compiler is critical to the degree of automation that can be attained. Moreover, knowledge-centric networks allow software developers an economy of scale in support of software reuse.

Published in Proceedings of the 28th Conference ICC&IE, 5-7 March 2001.

15. SUBJECT TERMS
expert compilers
expert systems
randomization
reuse
software engineering

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
U	U	U	UU	6

19a. NAME OF RESPONSIBLE PERSON
Dr. Stuart Rubin, D73C

19B. TELEPHONE NUMBER *(Include area code)*
(619) 553-3554

On the Role of Randomization in Software Engineering

Stuart H. Rubin, Ljiljana Trajkovic, James Boerke, and Robert J. Rush Jr.

Abstract – *Randomization is defined to mean the removal of redundancy from information. In this sense, it is synonymous with information compression; although, randomization may extend beyond syntactic representation to include domain-specific semantic elements as well. This paper serves to make clear the ubiquitous role assumed by randomization in all aspects of software engineering – from programming language design to program design to testing. It goes on to show that the representation of knowledge in what is termed an expert compiler is critical to the degree of automation that can be attained. Moreover, knowledge-centric networks allow software developers an economy of scale in support of software reuse.*

Keywords – expert compilers, expert systems, randomization, reuse, software engineering

Introduction

We consider the neat vs. the scruffy approach to an artificial intelligence as first proposed by Minsky. In particular, the brain has been modeled as a society of mind [1]. At some levels, the brain clearly processes information, at least in part, using a scruffy approach (e.g., vision). At other levels, the brain clearly processes information, at least in part, using a neat approach (e.g., logical reasoning). It has been proven by Lin and Vitter [2] that neural networks that have a hidden layer are NP-hard to train. Thus, a domain-specific representation is needed here for at least purposes of scalability. A domain-specific representation defines the abstract data types and overall structure of both programming language and program.

On Minimizing Entropy

We acknowledge the fundamental need for domain-specific knowledge in keeping with Rubin's proof of the Unsolvability of the Randomization Problem [3]. The question as to the degree to which a neural net, or system of networks, can learn to decrease its entropy may be reduced to the capability for forming domain-specific generalizations. This capability for domain-specific generalization goes well beyond the statistical boundaries of contemporary neural net generalization. A second application of the Unsolvability of the Randomization Problem reduces the capability for forming domain-specific generalizations to the capability for forming domain-specific representations. But then, this capability is dependent on previous structure. Thus, it is argued that the entire issue of a neat vs. scruffy approach is provably irrelevant – in agreement with Minsky. Instead, the brain must form domain-specific representations using evolutionary randomization (see below). All that matters is to minimize the entropy of the information-theoretic model using randomization. Of course, there is also a practical dimension that involves issues pertaining to spatial-temporal realization. However, even here the Unsolvability of the Randomization Problem implies the necessary search for ever-better domain-specific hardware.

On the Algorithm

We know that the central reason for the failure of the Japanese Fifth Generation project is that their system was built to be capable of deductive, but not inductive reasoning [4]. In other words, it followed a second order predicate calculus model. They learned that the logic must yield to the algorithm in all matters pertaining to creative or inductive reasoning. The quality of an algorithm, in turn, hinges on the degree to which its space-time image can be minimized through reuse. We note that any algorithm sufficiently complex to be capable of self-refe-

- S.H. Rubin, R.J. Rush, and James Boerke are with the Space and Naval Warfare Systems Center, San Diego, 53560 Hull Street, San Diego, CA 92152-5001, USA.
  Tel. (619) 553-3554; Fax (619) 553-1130
  E-mail srubin@spawar.navy.mil; rushr@spawar.navy.mil; jboerke@spawar.navy.mil
- L. Trajkovic is with Simon Fraser University, Vancouver, BC Canada
  Tel. (604) 291-3998; Fax (604) 291-4951; E-mail ljilja@cs.sfu.ca

rence can never be certified as bug free. This is a law of nature. Nevertheless, testing and scalability are maximized if reuse is maximized. Construction time is minimized if reuse is maximized. RISC chips are but one example of these principles applied to hardware design. Simply put, creativity is necessarily of an algorithmic nature and non-trivial algorithms and their associated platforms (i.e., of significant scale) cannot be had or run in the absence of first principles of randomization and reuse. If all this sounds a little foreign, it may help to recall that relativistic effects may also sound foreign to the average person due to the relatively high velocity of light with respect to their daily experiences.

On Problem Reduction

It has been shown (e.g., the Wizard neural system) that the triangle inequality applies to neural systems. For example, a neural system may learn to recognize a fork invariant of its position using c fundamental memories. However, if two neural systems are constructed and connected such that the first learns to rotate an object such as a knife to a normal position and the second then attempts to recognize this object, then the total number of fundamental memories needed to recognize the fork invariant of its position will be significantly less than c (i.e., the triangle inequality). The challenge is to scale such problem reductions. It is argued that an object-oriented society of mind model will serve to facilitate analysis. For example, if one neural network is trained on what a car is and a companion network is trained say on what a Toyota Corolla is (i.e., an instance of car), then not only is the entropy of the system decreased, but the reusability of the networks is proportionately increased.

Software Reuse and Retrieval

Software reuse facilitates the construction of ever-more complex programs. There are at least two key issues here. The first pertains to the level of the language of representation. Here, the higher the level, the more reusable the code. For example, compare the reusability of a program written in machine code with that written in some higher-level language. Clearly, the higher the level of the representation, the more amenable that representation is to reuse by way of modification. The capability for undergoing successful modification is also important because excepting certain GUI application components, software needs to be customizable.

Second, software modules need to be indexed for retrieval. This is similar to a case-based reuse mechanism (i.e., CBR); although, it differs in that there is no need to adapt the software as a case. Rather, we will argue that the software needs to be modifiable by substitutive transformation followed by expert compilation (see below).

It is convenient to think of the process of building software as being isomorphic to that of putting together a puzzle. One needs to retrieve just the right piece from the box using a necessarily vague description of the piece. Moreover, this puzzle is something of a fractal in that each piece can be recursively decomposed into sub-pieces until some primitive level is attained. Retrieval implies inspection (e.g., Are the pieces corners of the correct shape to fit?). In the software realm, such inspection can only be accomplished if the piece or component is expressed in a

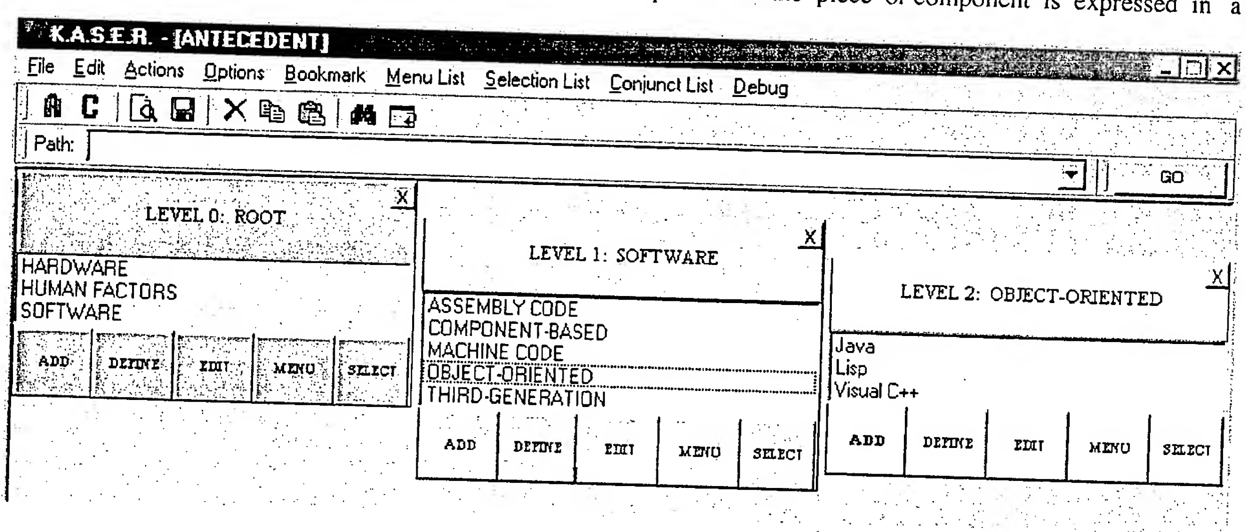

Fig. 1 A Hierarchical Sequence of Object Menus for Software Retrieval

sufficiently high-level language to be humanly understandable. We use a hierarchical sequence of object menus for retrieval as illustrated in Fig. 1.

```
((DEFUN MYSORT (S)
  (COND ((NULL S) NIL)
    (T (CONS (MYMIN S (CAR S)) (MYSORT (REMOVE (MYMIN S (CAR S)) S)))))))
? io
((((1 3 2)) (1 2 3)) (((3 2 1)) (1 2 3)) (((1 2 3)) (1 2 3)))
? (pprint (setq frepos '((CRISPY'
      (DEFUN MYSORT (S)
        (COND
         (FUZZY
          ((NULL S) NIL)
          ((ATOM (FUZZY S ((FUZZY CAR CDR) S))) NIL))
         (T (CONS (MYMIN S (CAR S))
         (MYSORT (REMOVE (MYMIN S (CAR S)) S))))))))))


((CRISPY '(DEFUN MYSORT (S)
   (COND (FUZZY ((NULL S) NIL) ((ATOM (FUZZY S ((FUZZY CAR CDR) S))) NIL))
    (T (CONS (MYMIN S (CAR S)) (MYSORT (REMOVE (MYMIN S (CAR S)) S))))))))

; Note that (ATOM S) was automatically programmed using the large fuzzy function space.

? (pprint (auto frepos io))

((DEFUN MYSORT (S)
  (COND ((ATOM S) NIL)
   (T (CONS (MYMIN S (CAR S)) (MYSORT (REMOVE (MYMIN S (CAR S)) S)))))))

; Note that each run may create syntactically different, but semantically equivalent
; functions:

? (pprint (auto frepos io))

((DEFUN MYSORT (S)
  (COND ((NULL S) NIL)
   (T (CONS (MYMIN S (CAR S)) (MYSORT (REMOVE (MYMIN S (CAR S)) S)))))))
```

Fig. 2 A Sample Fuzzy Program in an Extended Lisp

Software Debugging

In addition to software reuse and retrieval, software debugging also has strong ties to randomization. For example, consider the debugging of any sort program. It follows from the Unsolvability of the Randomization Problem [3] that any program sufficiently complex to be capable of self-reference can never be assured to be totally bug free. Rather, the best that can be done is to test it to satisfaction. There are two separate, but related issues here. First, how

does one develop a portfolio of test cases for maximal coverage and second, what principles does one employ to construct software so that it is relatively bug free?

Test cases should cover as many execution paths as practical; yet, be minimal in number so as to render the testing process practical. For example, when testing a sort program, one does not want test cases such as: (1) (2 1) (3 2 1) (4 3 2 1) … because the test vectors are mutually symmetric. Rather, one needs to develop an approximation of a basis (e.g., akin to the basis of a matrix). For example: () (1) (2 1) (1 3) (3 1 2) … Furthermore, Rubin [5] [6] has developed a fuzzy programming language that accepts a generalized program description and uses an orthogonal sequence of test cases to instantiate a working program, if possible. What is interesting here is that the induced program is no better nor worse than the supplied test sequence. This is testing in reverse and serves to underscore the critical role played by test case selection in determining the overall quality of the resulting software produced. One could argue that the aforementioned process works well for functional programming, but what about say for visual programming? How does one map test cases to form? The answer is that visual programming (e.g., creating GUIs) requires testing as does any other; although, here an expert or expert system is required to provide the feedback that determines the success or failure of each test. Notice that the process of randomization is pervasive: It exists in the generation of a minimal set of test cases; it exists in the specification of a fuzzy program (i.e., a minimal representation of a program space whose instances approximate useful working programs); it exists in the reuse of expert systems for GUI design; and, it surely exists in all related software processes too detailed to be described here. Fig. 2 is a depiction of a fuzzy program written in an extended version of the Lisp language [7] [8].

To construct software that is relatively bug-free, one needs to maximize the testability of every included software unit. This is necessary because, in keeping with the above arguments, software can only be certified in proportion to its having been tested using mutually random test cases that attempt to approximate the environment in which it will be used. For example, consider the development of the Fortran subroutine. Instead of transferring to different sections of possibly erroneous code, all these would-be calls are instead transferred to a parametized randomization of their semantics. The greater number of calls here serves to increase the number of tests made of the subroutine with the result that the overall quality of the program is improved through the use of subroutines. The same argument extends to the use of objects and components. Thus, we see that what may be termed, "randomized programming practices" results in improved code quality.

Expert Compilers

Expert compilers were first defined by Geoffrey Hindin [9]. Now, it is well-known that programmer productivity bears proportion to the level of the programming language in which a program may be expressed. Thus, we saw a six-fold jump in productivity in the 70s when assembly-language programming gave way to Fortran (i.e., 3d generation language) programming. Smaller gains are reported for object and component-based programming due to complexity issues. Simply put, the new problem in increasing programmer productivity is that 3d generation (universal) languages do not in effect substitute for user domain knowledge. One still needs to program every domain detail to get the program to work as desired. Expert compilers relax this stipulation by providing for rule-based knowledge insertion. For example, consider the following 3d generation vs. expert compiled program.

In Fig. 3, the expert rules act on the expert specification to produce the third generation code. The rules have been oversimplified for purposes of illustration. Notice that the expert specification is at a higher level and thus easier to debug and be more productive in through the use of the expert compiler. In fact, the separation of the domain knowledge from the program knowledge may be viewed as an extension of the traditional separation of the knowledge base from the inference engine in an expert system. Notice the parallels with the introduction of the Fortran subroutine described above. That is, an expert compiler is yet another form of randomization!

In traditional software engineering practice, an extensible language (e.g., Lisp) serves many of the same requirements for randomization. However, the difference between an extensible language and expert compilation is that the knowledge is embodied in the object in an extensible language, which tends to delimit its reusability. Such is not the case with an expert compiler. Now, as the size of the object gets smaller, its reusability increases until in the limit it offers the same advantages offered by an equivalent expert compiler, or so it would seem. What's missing from this argument is any notion of execution efficiency. That is, as the objects get smaller, the programs that they detail tend to loose efficiency.

Third Generation:

```
Repeat
  Read x, y;
Until
  y <> 0;
ratio = x/y;
If ratio > 0 then
  Print ratio
Else
  Print –ratio;
```

Expert Specification:

```
Read x, y;
ratio = x/y;
Print ratio;
```

Expert Rules:

```
If "/" Then denom <> 0
If "/" Then ratio +
```

Fig. 3 The Power of an Expert Compiler

An expert compiler can optimize code and thereby offers the better of two approaches. On the one hand, it frees the user to express a program in relatively simple and cognitively straightforward terms. On the other hand, the resulting sub-optimal program can then be automatically transformed into a more efficient form. For example, the typical computer scientist will find it far easier to write an $O(n^2)$ Bubble or Insertion sort than he/she will to write an $O(n \log n)$ Quicksort. They all may have the same test suite and the expert compiler can incrementally transform one into the other given an economy of scale. Note that the details pertaining how to accomplish this could occupy an entire volume. Thus, we necessarily concern ourselves with the concept for now.

A simple expert compiler can become more complex by way of fusing a network of domain-specific knowledge bases. Notice that as more and more domain-knowledge can be brought to bear on the compilation, the level of the effectively transformed language can increase. All this may be succinctly stated to be a consequence of randomization. Moreover, the language in which the rule predicates are represented in each rule base can also be subject to expert compilation. We call this a knowledge-based bootstrap. Observe that it too is recognizable as being a higher-level randomization. Informally, we call this a capability for the dynamic domain-specific representation of knowledge.

Any network of expert compilers can grow to be exceedingly complex. After all, given that there will be errors, how does one trace a resultant error back to its source? This problem can be accentuated through the use of asynchronous MIMD architectures. The solution is to keep the human in the loop. Moreover, it is key to note that the more reusable the representation of knowledge (and software), the greater will be the propagation of repairs. This follows because highly reusable components are invoked by many other components. Correcting one error then serves to correct many errors. Think of this as being the inverse of testing, where the higher the degree of reusability the more likely the program will be valid.

What emerges from the previous discussion is the notion that higher-level software is going to be more structured in all its salient aspects. Instead of being hand-crafted, it will be assembled – not by humans, but by machines that were programmed for its assembly. The central thesis of this section is that higher structures necessarily go beyond the lax bounds imposed by domain-independent representation. While such representations account for objects and components, they place the entire burden of assembly upon the user. Here, the knowledge source is a sole source; namely, the human. To climb above the third-generation plateau, one needs to capture domain-specific knowledge for reuse. Such is accomplished by an expert compiler, which effects in theoretical terms, semantic randomization. Note that fourth-generation languages may appear to get around this problem without resorting to expert compilation, but this is a convenient illusion. The reason for the illusion is that such languages are not universal. In a practical sense, this means that they are not flexible or conveniently extensible. Also, fifth generation languages (e.g., Lisp, Prolog, *et al.*) provide tools for the construction of expert compilers, but are not expert compiled per se.

Conclusions

Insight on whether or not the brain relies primarily upon a neural representation, a symbolic one, neither, or both will enable the construction of ever-more intelligent systems. Moreover, this paper suggests that a new view of software reuse will evolve in the form of a taxonomy:

- Level 5: Reuse requires a dynamic domain-specific representation of knowledge.
- Level 4: Reuse requires the application of knowledge bases (e.g., expert compilers).
- Level 3: Reuse occurs in the form of objects and components.
- Level 2: Reuse occurs at the code level and allows for parametization.
- Level 1: Reuse occurs at the level of immutable code.

If the brain is to follow this taxonomic description, then level 1 corresponds to declarative memory, level 2 to procedural memory, level 3 to physical creativity (e.g., substituting a barstool for a chair), level 4 to abstract creativity (e.g., substituting a table for a chair), and level 5 to *creative* creativity (e.g., who needs to sit anyway?).

The reader may be curious as to why the fusion of brain science with software engineering in this paper. After all this is like comparing the proverbial apples and oranges. The connection again lies in the information-theoretic term, "randomization". Practically speaking, what software lacks in structure in the sense made clear by this paper, the human brain must supply in the form of knowledge. Now, randomization theory holds that the human should supply novel knowledge exactly once (i.e., random input) and the machine extend that knowledge by way of capitalizing on domain symmetries (i.e., expert compilation). This means that in the future, programming will become more creative and less detailed and thus the cost per line of code will rapidly decrease. We have learned from various sources that the White House has chosen Lisp for programming some server-based applications. It is claimed that they experienced a 500 percent improvement in productivity as a result of the extensible features imbued in this language. Again, this success story does not begin to touch on the possibilities offered by networked expert compilers of scale. According to Bob Manning [7],

> Processing knowledge is abstract and dynamic. As future knowledge management applications attempt to mimic the human decision-making process, a language is needed which can provide developers with the tools to achieve these goals. Lisp enables programmers to provide a level of intelligence to knowledge management applications, thus enabling ongoing learning and adaptation similar to the actual thought patterns of the human mind.

In conclusion, the solution to the software bottleneck will be cracking the knowledge acquisition bottleneck in expert systems (compilers). We need to study knowledge representation and learning, rule-based compilers, and associated architectures. For example, it is possible that knowledge-based segments can be retrieved on demand over the Internet, which can provide the necessary economy of scale required for the successful implementation of networked expert compilers [10].

Acknowledgments

This paper includes the work of U.S. Government employees performed in the course of their employment and is not subject to copyright. It is approved for public release with an unlimited distribution. This work was supported in part by the Office of Naval Research, NSERC, and the BC Advanced Systems Institute Fellowship.

References

[1] M. Minsky, *The Society of Mind*, New York, NY: Simon and Schuster, Inc., 1987.

[2] J-H. Lin and J.S. Vitter, "Complexity Results on Learning by Neural Nets," *Machine Learning*, vol. 6, no. 3, pp. 211-230, 1991.

[3] S.H. Rubin, "Computing with Words," *IEEE Trans. Syst. Man, Cybern.*, vol. 29, no. 4, pp. 518-524, 1999.

[4] E.A. Feigenbaum and P. McCorduck, *The Fifth Generation*. Reading, MA: Addison-Wesley Publishing Co., 1983.

[5] S.H. Rubin, "A Fuzzy Approach Towards Inferential Data Mining," *Computers and Industrial Engineering*, vol. 35, nos. 1-2, pp. 267-270, 1998.

[6] S.H. Rubin, "A Heuristic Logic for Randomization in Fuzzy Mining," *J. Control and Intell. Systems*, vol. 27, no. 1, pp. 26-39, 1999.

[7] B. Manning, "Smarter Knowledge Management Applications: Lisp," *PC AI*, vol. 14, no. 4, pp. 28-31, 2000.

[8] E. Gat, "Lisp as an Alternative to Java," *Intelligence*, pp. 21-24, Winter 2000.

[9] J. Hindin, "Intelligent Tools Automate High-Level Language Programming," *Computer Design*, vol. 25, pp. 45-56, 1986.

[10] I. Ben-Shaul and G. Kaiser, "Coordinating Distributed Components over the Internet," *IEEE Internet Computing*, vol. 2, pp. 83-86, 1998.